DL

DL PRACT 1

**Cell 1: Importing Libraries**

python

CopyEdit

import tensorflow as tf

* **import tensorflow**: Loads the TensorFlow library, a deep learning framework.
* **as tf**: Aliases TensorFlow to tf for easier use in code.

python

CopyEdit

from sklearn.model\_selection import train\_test\_split

* **from sklearn.model\_selection**: Refers to the model\_selection module in scikit-learn.
* **import train\_test\_split**: Imports a function that splits data into training and testing sets.

python

CopyEdit

from sklearn.preprocessing import StandardScaler

* Imports StandardScaler, which standardizes features by removing the mean and scaling to unit variance.

python

CopyEdit

from sklearn import metrics

* Imports the metrics module to evaluate model performance (like accuracy, MSE, etc.).

python

CopyEdit

import pandas as pd

* Imports the pandas library for data manipulation, often used with tabular data.

python

CopyEdit

import numpy as np

* Imports NumPy for numerical operations (arrays, math functions).

python

CopyEdit

import matplotlib.pyplot as plt

* Imports pyplot from matplotlib for plotting graphs.

python

CopyEdit

import seaborn as sns

* Imports seaborn, a statistical visualization library based on matplotlib.

python

CopyEdit

%matplotlib inline

* A **Jupyter magic command** that ensures plots appear directly below the code cell that produced them.

python

CopyEdit

from tqdm.notebook import tqdm

* Imports tqdm for showing progress bars in loops, adapted to Jupyter notebook UI.

python

CopyEdit

import warnings

warnings.filterwarnings("ignore")

* Suppresses warning messages to keep notebook output clean.

**Cell 2: Load Dataset Reference**

python

CopyEdit

boston = tf.keras.datasets.boston\_housing

* Assigns the boston\_housing dataset from TensorFlow's dataset module to the variable boston.

**Cell 3: Inspect Object**

python

CopyEdit

dir(boston)

* **dir()**: Built-in function that lists the attributes and methods of an object—in this case, the boston dataset.

Would you like me to continue line-by-line for the rest of the notebook?

4o

**You said:**

yes

**ChatGPT said:**

Here’s a **line-by-line detailed explanation** of the remaining code cells:

**Cell 4: Read Dataset**

python

CopyEdit

df = pd.read\_csv("boston.csv")

* Reads a CSV file named boston.csv into a pandas DataFrame named df.

**Cell 5: Prepare Features and Labels**

python

CopyEdit

X = df.drop(columns=['MEDV', 'RAD'])

* Drops the 'MEDV' (target variable) and 'RAD' (often considered a categorical or less relevant feature) columns from the dataset.
* Stores the remaining columns (features) in X.

python

CopyEdit

Y = df['MEDV']

* Extracts the 'MEDV' column as the target/output variable Y.

**Cell 6: Split Dataset**

python

CopyEdit

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(X, Y, test\_size=0.2, random\_state=42)

* Splits X and Y into training and testing sets.
* 80% goes to training, 20% to testing.
* random\_state=42 ensures reproducibility.

**Cell 7: Check Shapes**

python

CopyEdit

X\_train.shape, Y\_train.shape, X\_test.shape, Y\_test.shape

* Returns the shape (rows, columns) of training and test sets for both features and labels.

**Cell 8: Initialize Scaler**

python

CopyEdit

scaler = StandardScaler()

* Creates a StandardScaler instance to standardize features.

**Cell 9: Scale Features**

python

CopyEdit

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

* fit\_transform: Computes mean and std on training data and scales it.
* transform: Uses the same mean and std to scale test data (prevents data leakage).

**Cell 10: Scale Labels**

python

CopyEdit

Y\_train\_scaled = scaler.fit\_transform(Y\_train.values.reshape(-1, 1))

Y\_test\_scaled = scaler.transform(Y\_test.values.reshape(-1, 1))

* Reshapes Y\_train/Y\_test to 2D arrays (required by scaler).
* Standardizes labels for better model convergence.

**Cell 11: Define Neural Network Model**

python

CopyEdit

model = tf.keras.models.Sequential([

tf.keras.layers.Input(shape=(11,), name='input-layer'),

tf.keras.layers.Dense(100, name='hidden-layer-2'),

tf.keras.layers.BatchNormalization(name='hidden-layer-3'),

tf.keras.layers.Dense(50, name='hidden-layer-4'),

tf.keras.layers.Dense(1, name='output-layer')

])

* Creates a Sequential model with:
  + Input layer with 11 features.
  + First dense (fully connected) layer with 100 neurons.
  + Batch normalization for stabilizing learning.
  + Second dense layer with 50 neurons.
  + Output layer with 1 neuron (regression output).

python

CopyEdit

tf.keras.utils.plot\_model(model, show\_shapes=True)

* Plots the architecture of the model, showing input/output shapes.

**Cell 12: Model Summary**

python

CopyEdit

model.summary()

* Displays the architecture, number of parameters, and layers.

**Cell 13: Compile Model**

python

CopyEdit

model.compile(

optimizer='adam',

loss='mse',

metrics=['mae']

)

* Compiles the model:
  + adam: adaptive learning rate optimizer.
  + mse: mean squared error as the loss.
  + mae: mean absolute error as an evaluation metric.

**Cell 14: Train Model**

python

CopyEdit

history = model.fit(X\_train, Y\_train, batch\_size=32, epochs=20,

validation\_data=(X\_test, Y\_test))

* Trains the model for 20 epochs with batch size of 32.
* Evaluates performance on validation (test) set after each epoch.

**Cell 15: Plot Training Metrics**

python

CopyEdit

pd.DataFrame(history.history).plot(figsize=(10,7))

plt.title("Metrics graph")

plt.show()

* Converts training history to a DataFrame.
* Plots the loss and metric (mae) over epochs.

**Cell 16: Predict on Test Data**

python

CopyEdit

y\_pred = model.predict(X\_test)

* Uses the trained model to predict housing prices on the test data.

**Cell 17: Visualize Predictions**

python

CopyEdit

sns.regplot(x=Y\_test, y=y\_pred)

plt.title("Regression Line for Predicted values")

plt.show()

* Plots a regression line comparing actual (Y\_test) and predicted values (y\_pred).
* Helps visualize prediction accuracy.

**Cell 18: Evaluate with Metrics**

python

CopyEdit

def regression\_metrics\_display(Y\_test, y\_pred):

print(f"MAE is {metrics.mean\_absolute\_error(Y\_test, y\_pred)}")

print(f"MSE is {metrics.mean\_squared\_error(Y\_test,y\_pred)}")

print(f"R2 score is {metrics.r2\_score(Y\_test, y\_pred)}")

regression\_metrics\_display(Y\_test, y\_pred)

* Defines and calls a function that prints:
  + **MAE**: Mean Absolute Error.
  + **MSE**: Mean Squared Error.
  + **R2 score**: Coefficient of determination (goodness of fit).

DL PARCTICAL 2

**Cell 4: Inspect Data**

python

CopyEdit

print(train\_df.head())

print(train\_df.columns)

* Displays the first few rows of train\_df and prints the column names.

**Cell 5: Extract Text and Labels**

python

CopyEdit

X\_train\_text = train\_df['text'].astype(str)

y\_train = train\_df['class']

X\_test\_text = test\_df['text'].astype(str)

y\_test = test\_df['class']

* Extracts:
  + Input features (text) as strings.
  + Labels (class) as targets.

**Cell 6: Tokenization and Padding**

python

CopyEdit

tokenizer = Tokenizer(num\_words=10000, oov\_token='<OOV>')

tokenizer.fit\_on\_texts(X\_train\_text)

X\_train\_seq = tokenizer.texts\_to\_sequences(X\_train\_text)

X\_test\_seq = tokenizer.texts\_to\_sequences(X\_test\_text)

max\_length = 100

X\_train = pad\_sequences(X\_train\_seq, maxlen=max\_length, padding='post')

X\_test = pad\_sequences(X\_test\_seq, maxlen=max\_length, padding='post')

* **Tokenizer**: Converts text to integer sequences, handling unknown words with <OOV>.
* **fit\_on\_texts**: Builds vocabulary from training data.
* **texts\_to\_sequences**: Converts sentences to lists of integers.
* **pad\_sequences**: Ensures all sequences are the same length (100) by padding.

**Cell 7: Build Model**

python

CopyEdit

model = Sequential([

Input(shape=(max\_length,)),

Embedding(input\_dim=10000, output\_dim=64),

GlobalAveragePooling1D(),

Dense(64, activation='relu'),

Dropout(0.2),

Dense(1, activation='sigmoid')

])

* **Sequential**: Linear stack of layers.
* **Input**: Accepts sequences of length 100.
* **Embedding**: Turns word indices into dense vectors.
* **GlobalAveragePooling1D**: Reduces each sequence to a single average vector.
* **Dense (64)**: Fully connected layer with ReLU activation.
* **Dropout**: Randomly drops 20% of neurons during training.
* **Dense (1)**: Output layer for binary classification (sigmoid).

**Cell 8: Compile Model**

python

CopyEdit

model.compile(loss='binary\_crossentropy',

optimizer='adam',

metrics=['accuracy'])

* **binary\_crossentropy**: Suitable for binary classification.
* **adam**: Adaptive learning optimizer.
* **accuracy**: Metric to evaluate performance.

**Cell 9: Model Summary**

python

CopyEdit

model.summary()

* Shows detailed model architecture, number of parameters per layer.

**Cell 10–11: Convert Data Types and Encode Labels**

python

CopyEdit

X\_train = np.array(X\_train, dtype='int32')

X\_test = np.array(X\_test, dtype='int32')

y\_train = np.array(y\_train, dtype='float32')

y\_test = np.array(y\_test, dtype='float32')

* Ensures all arrays have proper types for TensorFlow models.

python

CopyEdit

le = LabelEncoder()

y\_train = le.fit\_transform(y\_train)

y\_test = le.transform(y\_test)

* Encodes string labels (Positive, Negative) into integers (0 or 1).

python

CopyEdit

y\_train = y\_train.astype('float32')

y\_test = y\_test.astype('float32')

* Converts encoded labels to float, as expected by Keras.

**Cell 12: Train Model**

python

CopyEdit

history = model.fit(

X\_train, y\_train,

epochs=30,

batch\_size=32,

validation\_data=(X\_test, y\_test)

)

* Trains the model for 30 epochs using batches of 32.
* Validates on test set during training.

**Cell 13–14: Evaluate Model**

python

CopyEdit

loss, accuracy = model.evaluate(X\_test, y\_test)

print(f"Test Accuracy: {accuracy:.2f}")

* Evaluates model performance on test data.

python

CopyEdit

loss, accuracy = model.evaluate(X\_train, y\_train)

print(f"Train Accuracy: {accuracy:.2f}")

* Evaluates performance on training data (check for overfitting).

**Cell 15: Visualize Training History**

python

CopyEdit

pd.DataFrame(history.history).plot(figsize=(10,7))

plt.title("Metrics Graph")

plt.show()

* Plots training/validation loss and accuracy over epochs.

**Cell 16–17: Predict and Report**

python

CopyEdit

y\_pred\_probs = model.predict(X\_test)

y\_pred = (y\_pred\_probs > 0.5).astype('int32')

* Predicts probabilities.
* Converts probabilities to binary predictions (threshold = 0.5).

python

CopyEdit

print(classification\_report(y\_test, y\_pred))

* Prints precision, recall, f1-score, and accuracy.

**Cell 18: Confusion Matrix**

python

CopyEdit

cm = confusion\_matrix(y\_test, y\_pred)

plt.figure(figsize=(5,4))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',

xticklabels=['Predicted Neg', 'Predicted Pos'],

yticklabels=['Actual Neg', 'Actual Pos'])

plt.xlabel('Predicted')

plt.ylabel('Actual')

plt.title('Confusion Matrix')

plt.show()

* Visualizes confusion matrix to show true vs predicted class distribution.

**Cell 19–20: Predict on New Samples**

python

CopyEdit

sample = ["<sample\_text>"]

sample\_seq = tokenizer.texts\_to\_sequences(sample)

sample\_pad = pad\_sequences(sample\_seq, maxlen=max\_length, padding='post')

prediction = model.predict(sample\_pad)

print("Positive" if prediction[0][0] > 0.5 else "Negative")

* Preprocesses a new sample the same way as training data.
* Predicts sentiment: prints "Positive" if predicted probability > 0.5, else "Negative".

DL PARCTICAL 3

Here’s a **complete line-by-line explanation with detailed syntax** of your code, which builds and evaluates a CNN model on image data like Fashion MNIST:

**Load Training and Test Data**

y\_train = train\_df.iloc[:, 0].values

* Selects the **first column** (label/class) from train\_df.
* .iloc[:, 0]: selects all rows (:) and column at index 0.
* .values: converts to NumPy array.

X\_train = train\_df.iloc[:, 1:].values

* Selects all columns **except the first one** (the pixel values) for training inputs.

y\_test = test\_df.iloc[:, 0].values

X\_test = test\_df.iloc[:, 1:].values

* Same process for the test dataset.

**Normalize Pixel Values**

X\_train = X\_train / 255.0

X\_test = X\_test / 255.0

* Pixel values range from 0 to 255.
* Dividing by 255.0 scales them to [0, 1] — better for neural networks.

**Reshape Data to Match CNN Input**

X\_train = X\_train.reshape(-1, 28, 28, 1)

X\_test = X\_test.reshape(-1, 28, 28, 1)

* Reshapes flat pixel arrays to 28×28 images with 1 color channel (grayscale).
* -1: Automatically infers the number of samples.

**🧠 Build the CNN Model**

model = tf.keras.Sequential([

tf.keras.layers.Conv2D(32, (3, 3), activation='relu', input\_shape=(28, 28, 1)),

* Conv2D(32, (3, 3)): 32 filters of size 3×3.
* activation='relu': introduces non-linearity.
* input\_shape=(28, 28, 1): input image dimensions.

tf.keras.layers.MaxPooling2D(2, 2),

* Downsamples feature maps by a 2×2 window.

tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),

tf.keras.layers.MaxPooling2D(2, 2),

* Another convolution and pooling layer pair.

tf.keras.layers.Flatten(),

* Flattens the 2D feature maps into a 1D vector for the dense layers.

tf.keras.layers.Dense(128, activation='relu'),

* Fully connected hidden layer with 128 units.

tf.keras.layers.Dropout(0.3),

* Prevents overfitting by randomly turning off 30% of neurons during training.

tf.keras.layers.Dense(10, activation='softmax') # 10 classes

])

* Output layer with 10 neurons (one per class).
* softmax: converts outputs to probabilities.

**Compile the Model**

model.compile(optimizer='adam',

loss='sparse\_categorical\_crossentropy',

metrics=['accuracy'])

* adam: adaptive optimizer.
* sparse\_categorical\_crossentropy: for multi-class classification **with integer labels**.
* metrics=['accuracy']: track accuracy during training.

**Train the Model**

history = model.fit(X\_train, y\_train, epochs=10, validation\_data=(X\_test, y\_test))

* Trains the model for 10 epochs.
* validation\_data: evaluates performance on test set after each epoch.
* history: stores accuracy/loss per epoch.

**Plot Training Metrics**

plt.plot(history.history['accuracy'], label='Train Accuracy')

plt.plot(history.history['val\_accuracy'], label='Val Accuracy')

plt.plot(history.history['loss'], label='Train Loss')

plt.plot(history.history['val\_loss'], label='Val Loss')

plt.legend()

plt.title('Training History')

plt.show()

* Plots training & validation accuracy/loss.
* Useful for diagnosing overfitting or underfitting.

**Make Predictions and Evaluate**

y\_pred = np.argmax(model.predict(X\_test), axis=1)

* Predicts probability for each class on X\_test.
* np.argmax(..., axis=1): gets the class with the highest probability.

print(classification\_report(y\_test, y\_pred))

* Prints precision, recall, F1-score, and accuracy for each class.

cm = confusion\_matrix(y\_test, y\_pred)

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')

plt.xlabel("Predicted")

plt.ylabel("Actual")

plt.title("Confusion Matrix")

plt.show()

* Shows a **confusion matrix** heatmap.
* Helps identify which classes are confused with each other.

**Class Names for Display**

class\_names = [

"T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",

"Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"

]

* Maps class indices (0–9) to human-readable labels.

**Visualize Predictions on Random Images**

images = []

labels = []

random\_indices = random.sample(range(len(X\_test)), 10)

* Randomly selects 10 test images using random.sample.

for idx in random\_indices:

images.append(X\_test[idx])

labels.append(y\_test[idx])

images = np.array(images)

labels = np.array(labels)

* Collects the selected images and their true labels.

**Display Sample Predictions**

fig = plt.figure(figsize=(20, 8))

rows = 2

cols = 5

* Sets up a 2×5 grid for plotting 10 images.

for i, (image, true\_label) in enumerate(zip(images, labels), start=1):

ax = fig.add\_subplot(rows, cols, i)

prediction = model.predict(np.expand\_dims(image, axis=0), verbose=0)

predicted\_label = np.argmax(prediction)

ax.imshow(image.reshape(28, 28), cmap="gray")

ax.set\_title(f"Label: {class\_names[true\_label]}\nPred: {class\_names[predicted\_label]}")

ax.axis("off")

* For each selected image:
  + Predicts label.
  + Displays image with true and predicted label.

plt.tight\_layout()

plt.show()

* Adjusts subplot spacing and shows the plot.

Would you like a PDF version of this explanation for offline reference?

HPC

HPC1

Absolutely! Below is a **line-by-line explanation** of the given C++ code, especially focusing on **C++ syntax** and its relevance in **high performance computing (HPC)** using **OpenMP for parallelism**.

#include <iostream>

* **Includes** the standard C++ input-output stream library.
* Needed for using cout, cin, etc.

#include <vector>

* **Includes** the vector container from the C++ Standard Template Library (STL).
* vector is a dynamic array that can resize itself and is widely used in graph representations.

#include <queue>

* Includes the queue container from STL.
* queue is used in the BFS traversal (FIFO structure).

#include <omp.h>

* Includes OpenMP (Open Multi-Processing) library.
* Enables **parallel programming**, which is key in **HPC** to utilize multiple CPU cores.

using namespace std;

* Avoids writing std:: before STL classes and functions like cout, vector, etc.

**Graph Structure Definition**

struct Graph {

* Defines a struct called Graph. A struct is like a class, mainly used for grouping variables and functions.

int V;

* V is the number of vertices in the graph.

vector<vector<int>> adj;

* adj is an **adjacency list**: a vector of vectors to store connections (edges) for each node.
* Each adj[u] is a list of nodes connected to node u.

Graph(int V) : V(V), adj(V) {}

* **Constructor**: initializes the graph with V vertices.
* adj(V) creates V empty vectors, one for each vertex.

**Add Edge**

void addEdge(int u, int v) {

adj[u].push\_back(v);

adj[v].push\_back(u);

}

* Adds an **undirected edge** between u and v by adding each to the other’s list.

**BFS (Breadth-First Search)**

void BFS(int start) {

* Begins BFS traversal from the given start node.

vector<bool> visited(V, false);

* Keeps track of visited nodes to avoid cycles.

queue<int> q;

* FIFO queue for BFS traversal.

visited[start] = true;

q.push(start);

* Mark start as visited and enqueue it.

while (!q.empty()) {

int u = q.front(); q.pop();

cout << u << " ";

* While queue isn't empty, dequeue a node u and print it.

#pragma omp parallel for

for (int i = 0; i < adj[u].size(); i++) {

int v = adj[u][i];

if (!visited[v]) {

#pragma omp critical

{ visited[v] = true; q.push(v); }

}

}

* **Parallel loop using OpenMP**: tries to explore neighbors in parallel.
* #pragma omp critical protects shared resources (visited, q) from race conditions.
* ❗ **However**: Using visited[v] this way is not thread-safe unless properly synchronized (parallel BFS is complex and usually not written this way).

**DFS (Depth-First Search)**

void DFS(int start) {

vector<bool> visited(V, false);

* Similar visited array.

#pragma omp parallel

#pragma omp single nowait

DFSUtil(start, visited);

cout << endl;

}

* OpenMP starts a **parallel region**, but only **one thread** calls DFSUtil initially (single).
* Inside DFSUtil, other threads may help due to parallel loop.

void DFSUtil(int u, vector<bool>& visited) {

visited[u] = true;

cout << u << " ";

* Marks node u as visited and prints it.

#pragma omp parallel for

for (int i = 0; i < adj[u].size(); i++) {

int v = adj[u][i];

if (!visited[v]) DFSUtil(v, visited);

}

}

* **Attempts** to call recursive DFS in parallel for each neighbor.
* ⚠ But recursion and shared data (visited) without synchronization can cause **data races**.
* Better performance might require locking or task-based parallelism.

**Main Function**

int main() {

int V, edgeCount, u, v;

* Declare variables: V for vertices, edgeCount for number of edges, u and v for input.

cout << "Enter the number of vertices: ";

cin >> V;

Graph g(V);

* Read V and initialize the graph.

cout << "Enter the number of edges: ";

cin >> edgeCount;

cout << "Enter the edges (in format 'source destination'): \n";

for (int i = 0; i < edgeCount; i++) {

cin >> u >> v;

g.addEdge(u, v);

}

* Reads each edge and adds to the graph.

cout << "BFS traversal starting from node 0: ";

g.BFS(0);

cout << "DFS traversal starting from node 0: ";

g.DFS(0);

* Executes and prints the BFS and DFS traversals from node 0.

return 0;

}

* End of the program.

**Summary for High Performance Computing (HPC):**

* **OpenMP** is used to parallelize loops (#pragma omp parallel for) and tasks (#pragma omp single).
* However, proper **synchronization** (like #pragma omp critical) is essential to avoid race conditions when threads access shared variables (visited, q).
* **Parallel BFS and DFS** are complex and often need task-based models or locking mechanisms.

Would you like me to refactor this code for better parallel performance and thread safety?

HPC 2

**Parallel Bubble Sort**

cpp

CopyEdit

void bubbleSort(vector<int>& arr) {

int n = arr.size();

bool swapped = true;

* Function takes reference to a vector.
* swapped checks if any elements were swapped in a pass (loop continues until sorted).

cpp

CopyEdit

while (swapped) {

swapped = false;

* Outer loop continues until no swaps occur, meaning the array is sorted.

cpp

CopyEdit

#pragma omp parallel for

for (int i = 0; i < n - 1; i++) {

if (arr[i] > arr[i + 1]) {

swap(arr[i], arr[i + 1]);

swapped = true;

}

}

* **Incorrect parallelism** here:
  + swap and swapped = true can cause **race conditions**.
  + Bubble Sort is inherently sequential due to data dependencies (arr[i] relies on arr[i+1] from the previous state).
  + Also, swapped is a shared variable, modified by multiple threads—unsafe unless protected.
* ✅ **Key HPC note**: Bubble Sort is **not suitable** for parallelization due to dependencies between adjacent elements.

**Merge Function**

cpp

CopyEdit

void merge(vector<int>& arr, int l, int m, int r) {

vector<int> temp;

int left = l, right = m + 1;

* Merges two sorted halves: arr[l..m] and arr[m+1..r].

cpp

CopyEdit

while (left <= m && right <= r) {

if (arr[left] <= arr[right]) temp.push\_back(arr[left++]);

else temp.push\_back(arr[right++]);

}

while (left <= m) temp.push\_back(arr[left++]);

while (right <= r) temp.push\_back(arr[right++]);

* Classic merge logic for sorted arrays.

cpp

CopyEdit

for (int i = l; i <= r; i++) arr[i] = temp[i - l];

}

* Copies sorted result back to original array.

**Parallel Merge Sort**

cpp

CopyEdit

void mergeSort(vector<int>& arr, int l, int r) {

if (l < r) {

int m = l + (r - l) / 2;

* Recursively divides the array into halves.

cpp

CopyEdit

#pragma omp parallel sections

{

#pragma omp section

mergeSort(arr, l, m);

#pragma omp section

mergeSort(arr, m + 1, r);

}

* Runs two recursive calls **in parallel** using parallel sections.
* This is an effective way to parallelize divide-and-conquer algorithms.

cpp

CopyEdit

merge(arr, l, m, r);

}

}

* After recursive sort, merge the two sorted halves.

✅ **HPC Note**: Merge Sort is a good candidate for parallelization because:

* Divide steps can run independently.
* Merge is sequential but relatively fast.

**Main Function**

cpp

CopyEdit

int main() {

int n;

cout << "Enter the number of elements: ";

cin >> n;

* Read number of elements from user.

cpp

CopyEdit

vector<int> arr(n);

cout << "Enter the elements: ";

for (int i = 0; i < n; i++) cin >> arr[i];

* Read array elements.

**Timing and Sorting**

cpp

CopyEdit

clock\_t bubbleStart = clock();

bubbleSort(arr);

clock\_t bubbleEnd = clock();

* Measures time taken by bubbleSort.

cpp

CopyEdit

cout << "Sorted array using Bubble Sort: ";

for (int num : arr) cout << num << " ";

cout << endl;

* Displays sorted array.

cpp

CopyEdit

clock\_t mergeStart = clock();

mergeSort(arr, 0, n - 1);

clock\_t mergeEnd = clock();

* Measures time for mergeSort.

cpp

CopyEdit

cout << "Sorted array using Merge Sort: ";

for (int num : arr) cout << num << " ";

cout << endl;

**Timing Output**

cpp

CopyEdit

double bubbleDuration = double(bubbleEnd - bubbleStart) / CLOCKS\_PER\_SEC;

double mergeDuration = double(mergeEnd - mergeStart) / CLOCKS\_PER\_SEC;

* Calculates time in seconds.

cpp

CopyEdit

cout << "Bubble sort time in seconds: " << bubbleDuration << endl;

cout << "Merge sort time in seconds: " << mergeDuration << endl;

return 0;

}

* Outputs the time and ends the program.

**⚠️ Critical Issues & HPC Advice:**

**🔴 Bubble Sort:**

* **Not parallel-safe** in this implementation.
* Causes **race conditions** on swapped and on adjacent arr[i], arr[i+1].
* In practice: avoid using Bubble Sort in HPC.

**✅ Merge Sort:**

* Efficiently parallelized using #pragma omp sections.
* Could improve further by limiting parallelism to top levels to avoid excessive thread spawning.

HPC 3

Absolutely! Let’s **go step-by-step** and explain **everything in simple terms**, assuming you’re **new to C++** and learning about **parallel programming with OpenMP** for **high performance computing (HPC)**.

**🧠 What This Program Does**

It takes a list of numbers from the user, and calculates:

* ✅ Minimum value
* ✅ Maximum value
* ✅ Sum of all values
* ✅ Average of all values

And it does this **in parallel**, meaning **multiple CPU threads** work together to make it faster on large inputs.

**📦 Required Libraries**

#include <iostream> // To print and read input/output

#include <vector> // To use vectors (like lists in Python)

#include <omp.h> // OpenMP library for parallel programming

* #include adds a **library** or **header file** to your code.
* using namespace std; means we don’t have to write std:: everywhere.

**📘 Vectors in C++**

vector<int> vec(n);

* A vector is a **resizable array** (like Python's list).
* You use it to store n integers.

**🔁 Parallel Sum — Explained**

Here’s the fixed version:

int parallelSum(const vector<int>& vec) {

int sum = 0;

#pragma omp parallel for reduction(+:sum)

for (int i = 0; i < vec.size(); i++) {

sum += vec[i];

}

return sum;

}

**Let’s break this down:**

**1. const vector<int>& vec**

* const means we are not going to **change** this vector.
* & means we’re **passing by reference**, so it’s faster (avoids copying the whole list).

**2. sum = 0**

We start with sum = 0, just like a normal loop.

**3. #pragma omp parallel for reduction(+:sum)**

This tells the compiler:

* “Run this for loop **in parallel** using OpenMP.”
* reduction(+:sum) means:
  + Each thread makes its **own private copy** of sum.
  + At the end, OpenMP **adds them all together** safely.

**4. sum += vec[i]**

Each thread adds the elements it is assigned.

**🔻 Parallel Min (Smallest Number)**

int parallelMin(const vector<int>& vec) {

int min\_val = vec[0];

#pragma omp parallel for reduction(min:min\_val)

for (int i = 1; i < vec.size(); i++) {

if (vec[i] < min\_val) min\_val = vec[i];

}

return min\_val;

}

**🔎 What’s new here?**

* reduction(min:min\_val) tells OpenMP:
  + “Each thread keeps track of the smallest number it sees.”
  + At the end, OpenMP gives you the **smallest overall**.

**🔺 Parallel Max (Largest Number)**

Same idea, but for maximum:

int parallelMax(const vector<int>& vec) {

int max\_val = vec[0];

#pragma omp parallel for reduction(max:max\_val)

for (int i = 1; i < vec.size(); i++) {

if (vec[i] > max\_val) max\_val = vec[i];

}

return max\_val;

}

**🧮 Parallel Average**

float parallelAverage(const vector<int>& vec) {

return float(parallelSum(vec)) / vec.size();

}

* Calls the parallelSum() function.
* Divides the result by the number of elements.
* float(...) turns it into a decimal value.

**🧑‍💻 Main Function**

int main() {

int n;

cout << "Enter the number of elements: ";

cin >> n;

* This part asks the user how many numbers they will enter.

vector<int> vec(n);

cout << "Enter the elements: ";

for (int i = 0; i < n; ++i) cin >> vec[i];

* Creates a vector with size n.
* Asks the user to type n numbers, one after another.

**🖨️ Output Results**

cout << "Minimum value: " << parallelMin(vec) << endl;

cout << "Maximum value: " << parallelMax(vec) << endl;

cout << "Sum of values: " << parallelSum(vec) << endl;

cout << "Average of values: " << parallelAverage(vec) << endl;

return 0;

}

* Prints out the calculated min, max, sum, and average using your parallel functions.

**🚀 What Makes This High Performance?**

* **OpenMP** allows us to use **multiple CPU threads** at the same time.
* On large vectors, this can make your program **much faster**.
* We use **reduction** to avoid errors and safely combine results from different threads.

**📌 Final Notes**

* ✅ This is **correct, safe**, and performs better on **large datasets**.
* ⚠️ You need to **compile with OpenMP** support:

**Use this to compile:**

g++ yourfile.cpp -fopenmp -o yourprogram

* Then run:
* ./yourprogram

Would you like me to show how to **measure the time taken** by each of these operations using timers?

CUDA

Great! Let's walk through your full CUDA program **line-by-line** and explain it in **simple beginner-friendly terms**, focusing on both C++ and GPU (CUDA) concepts relevant to **high-performance computing**.

**🚀 Overview**

This program performs:

1. **Vector Addition** (A + B = C) using a GPU.
2. **Matrix Multiplication** (D × E = F) using a GPU.
3. Uses **CUDA kernels** for parallelism.

**✅ HEADERS**

#include <stdio.h> // For printf()

#include <stdlib.h> // For rand()

#include <cuda\_runtime.h> // CUDA API functions

These are required to:

* Print output
* Generate random numbers
* Use CUDA functions like cudaMalloc, cudaMemcpy, etc.

**🧮 Constants**

#define N 4

#define MATRIX\_SIZE 4

These define:

* N: number of elements in the vector
* MATRIX\_SIZE: 4x4 matrix

**🔧 Vector Addition Kernel**

\_\_global\_\_ void add(int\* A, int\* B, int\* C, int size) {

int tid = blockIdx.x \* blockDim.x + threadIdx.x;

if (tid < size) {

C[tid] = A[tid] + B[tid];

}

}

* \_\_global\_\_: Declares a **GPU function** (kernel).
* blockIdx, blockDim, threadIdx: CUDA built-in variables.
* tid is the **global thread index**.
* Each thread handles **one element** of the vector.

**🔧 Matrix Multiplication Kernel**

\_\_global\_\_ void multiply(int\* A, int\* B, int\* C, int size) {

int row = blockIdx.y \* blockDim.y + threadIdx.y;

int col = blockIdx.x \* blockDim.x + threadIdx.x;

* 2D block/grid setup for matrix.
* Each thread computes one (row, col) of result matrix.

int sum = 0;

if (row < size && col < size) {

for (int i = 0; i < size; ++i) {

sum += A[row \* size + i] \* B[i \* size + col];

}

C[row \* size + col] = sum;

}

}

* This is the **classic matrix multiplication** formula:
  + C[row][col] = A[row][i] \* B[i][col] summed over i.

**📤 Print Functions**

void printVector(const char\* name, int\* vec, int size)

void printMatrix(const char\* name, int\* mat, int size)

These just print the arrays in readable form.

**🚀 main() — Host Code**

**🧪 Vector Setup and GPU Allocation**

int A[N], B[N], C[N];

for (int i = 0; i < N; ++i) {

A[i] = rand() % 10;

B[i] = rand() % 10;

}

* Creates and fills two arrays (A and B) with random numbers.

int \*d\_A, \*d\_B, \*d\_C;

cudaMalloc(&d\_A, N \* sizeof(int)); // Allocates GPU memory

cudaMemcpy(d\_A, A, N \* sizeof(int), cudaMemcpyHostToDevice);

* Allocates memory on GPU (cudaMalloc) and copies data from CPU to GPU (cudaMemcpy).

**▶️ Launch Vector Add Kernel**

add<<<1, N>>>(d\_A, d\_B, d\_C, N);

cudaDeviceSynchronize(); // Wait for kernel to finish

* <<<1, N>>> launches 1 block with N threads.
* Each thread computes C[i] = A[i] + B[i].

cudaMemcpy(C, d\_C, N \* sizeof(int), cudaMemcpyDeviceToHost);

* Copies result C back to CPU from GPU.

**🧪 Matrix Setup and GPU Allocation**

int D[elements], E[elements], F[elements];

* elements = 16 (4x4 matrix).
* Fills D and E with random numbers.

**▶️ Launch Matrix Multiply Kernel**

dim3 threadsPerBlock(2, 2); // 2x2 threads per block

dim3 blocksPerGrid((size + 1)/2, (size + 1)/2);

* This gives enough blocks to cover the 4x4 matrix.
* multiply<<<blocksPerGrid, threadsPerBlock>>>(...) launches the matrix multiplication.

**✅ Cleanup and Result Display**

cudaFree(d\_A); cudaFree(d\_B); cudaFree(d\_C); // free memory

Always free GPU memory after use.

**🧠 Summary of Concepts**

| **Concept** | **Explanation** |
| --- | --- |
| \_\_global\_\_ | Declares a kernel (runs on GPU) |
| cudaMalloc | Allocates memory on GPU |
| cudaMemcpy | Copies memory between CPU and GPU |
| <<<blocks, threads>>> | Launch configuration for CUDA kernels |
| cudaDeviceSynchronize() | Wait until GPU finishes work |
| dim3 | Used to define 2D/3D thread and block sizes |
| blockIdx, threadIdx | Built-in CUDA variables for indexing |

**✅ How to Compile and Run**

nvcc cuda\_fixed.cu -o cuda\_fixed

./cuda\_fixed

nvcc is the NVIDIA CUDA compiler.

Would you like me to help you:

* Visualize the grid/thread setup?
* Add timing to measure GPU speedup?
* Or debug if you’re running into an error?

Let me know!